2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

# GPUMech: GPU Performance Modeling Technique based on Interval Analysis

Jen-Cheng Huang[1], Joo Hwan Lee[2], Hyesoon Kim[2], and Hsien-Hsin S. Lee[1]

[1]School of Electrical and Computer Engineering
[2]School of Computer Science
Georgia Institute of Technology
{*jencheng.huang, joohwan.lee, hyesoon.kim, leehs*}*@gatech.edu*

***Abstract*— GPU has become a first-order computing platform. Nonetheless, not many performance modeling techniques have been developed for architecture studies. Several GPU analytical performance models have been proposed, but they mostly target application optimizations rather than the study of different architecture design options.**

**Interval analysis is a relatively accurate performance modeling technique, which traverses the instruction trace and uses functional simulators, e.g., cache simulator, to track the stall events that cause performance loss. It shows hundred times of speedup compared to detailed timing simulations and better accuracy compared to pure analytical models. However, previous techniques are limited to CPUs and not applicable to multithreaded architectures.**

**In this work, we propose *GPUMech*, an interval analysis-based performance modeling technique for GPU architectures. GPUMech models multithreading and resource contentions caused by memory divergence. We compare GPUMech with a detailed timing simulator and show that on average, GPUMech has 13.2% error for modeling the round-robin scheduling policy and 14.0% error for modeling the greedy-then-oldest policy while achieving a 97x faster simulation speed. In addition, GPUMech generates CPI stacks, which help hardware/software developers to visualize performance bottlenecks of a kernel.**

***Keywords*-performance modeling; GPGPU; interval analysis; simulation;**

## I. Introduction

Performance models are attractive solutions to speed up simulation speed, especially for exploring early-stage design options. The models can provide greater insight for understanding architectures and applications. For example, analytical models have been proposed for both CPUs and GPUs [1], [4], [6], [9]. However, because of the relatively short history of GPU architectures, fewer analytical models and fast-simulation methods have been developed for GPUs.

Even though several GPU performance models are proposed [16], [2], [27], [24], unfortunately, most of them have focused on helping software optimizations. These models use hardware performance counters or program analysis to understand the performance bottlenecks of software, but they have not been used to explore different GPU architecture design options. Jia et al. [17] presented a regression-based GPU model that can explore the design space with various hardware parameters, but this model lacks in providing insight into performance changes.

Analytical models are simple and fast enough to get a first-order performance estimation, but they often have higher errors than detailed timing simulations. As an alternative solution to traditional analytical models, interval analysis was proposed [19], [13], which uses both trace-driven functional simulators and an analytical model to estimate core-level performance. The key difference between traditional analytical models and interval analysis is that while traditional analytical models use the average (or total summation) value of events, interval analysis traverses the instruction trace and tracks the performance degradation events, such as cache misses and branch mispredictions, and then a model is used to estimate the performance impact of each event. Tracking the events only requires functional simulations, e.g., cache simulation, which results in at least a 100x speedup compared to cycle-level simulations [10], [18], [5]. Hence, the interval analysis technique is faster than detailed cycle-level simulation while having a higher accuracy than traditional analytical models. Furthermore, interval analysis also provides the *CPI stack*, which shows the breakdown of different types of performance loss events and can provide an insight into performance behavior.

Nonetheless, no previous interval analysis technique has been proposed for GPU architectures. Prior interval analysis techniques cannot be naïvely applied to GPU architectures for two reasons. First, multithreading is not considered. Techniques have been developed for single-core or multi-cores in which only one thread executes per core but not for multithreaded architectures in which multiple threads concurrently execute to hide the memory latency. Second, control and memory divergence are not considered, since they are unique behaviors in SIMD/SIMT architectures.

To overcome these limitations, we propose an interval analysis technique, called *GPUMech*, the first interval analysis technique for GPU architectures. GPUMech profiles the instruction trace of every warp and uses a clustering algorithm to identify the representative warp. This is critical for kernels that have control-flow divergent warps. To model multithreading, GPUMech introduces the concept of *non-overlapped instructions* to accurately model the latency hiding capability of a scheduling policy. To handle performance degradations resulting from memory divergence, the resource contention in the memory system is also modeled.

The contributions of this paper are as follows.

- We propose the first interval analysis technique for GPU architectures, which can be also applied to other muli-

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.59

tithreaded architectures as well.

- We improve the interval analysis technique by modeling multithreading (Section IV-A) and resource contention of memory system. (Section IV-B).
- By leveraging our proposed technique, we propose the first CPI-stack visualization tool for GPU architectures to provide insights into performance bottlenecks. (Section VII).

## II. Background and Motivation

### A. Interval Analysis

The foundation of interval analysis was proposed by Karkhanis et al. [19] and Eyerman et al. [13]. The basic idea is that the performance of a processor is equal to the issue rate of a processor (a sustained performance) unless disruptive miss events occur such as branch mispredictions or cache misses. Performance is then estimated by subtracting the stall cycles due to different stall events from the maximum issue rate. Figure 1 illustrates interval analysis. An *interval* is defined as a sequence of instructions with the maximum issue rate followed by stall cycles. Functional simulators are used to detect stall events.

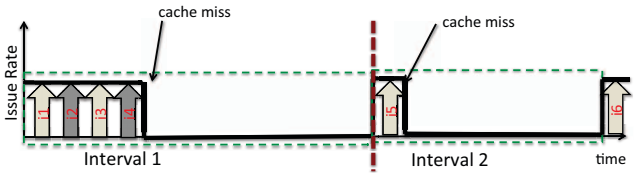



Figure 1. Example of interval analysis. (i: instruction.)

### B. Motivation

**Multithreading**: To apply CPU-based interval analysis to an in-order multithreaded architecture, e.g., GPU, several challenges exists. First, the previous interval analysis techniques have only been developed for single-threaded applications. Extending the model from a single-threaded architecture to a multithreaded architecture requires emulating instruction scheduling from multiple threads, which could incur high overhead.

$$\text{IPC}_{\text{core}} = \text{IPC}_{\text{single-warp\_performance}} \times \#\text{warps} \quad (1)$$

Alternatively, as shown in Eq. 1, a naïve approach to predict the performance of a multithreaded architecture is to use interval analysis for a single warp, which is then multiplied by the number of warps to get the final performance of multithreading. It assumes that the total cycles of a single warp remain unchanged while all instructions from the other warps can be issued during the stall cycles of the single warp to hide the stall cycles. However, this approach does not accurately model the warp scheduling policy, in which case the total cycles may change since *not all instructions from the other warps can overlap with the stall cycles*. The extra cycles, which are needed to issue the instructions that do not overlap with stall cycles, result in performance loss.

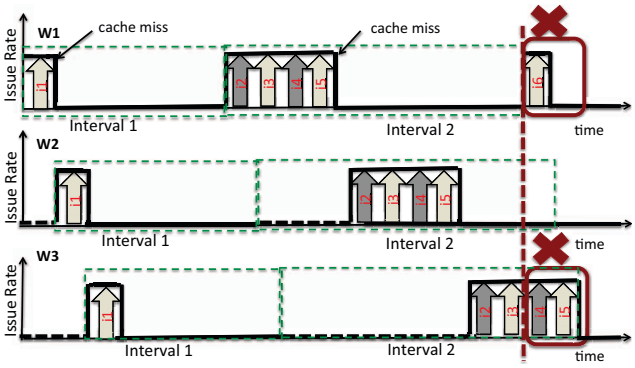



Figure 2. The case of interval analysis with multiple warps. (W: warp, i: instruction.)

Figure 2 shows an example. Assume that 3 warps are running on a core and each of which has two intervals: Interval 1 has 1 instruction while Interval 2 has 4 instructions. Both intervals have 10 stall cycles. The issue rate is 1 instruction per cycle. During interval 1, a total of three instructions can be issued, so the IPC for the core is 3/11, which is the same result as using Eq. 1 ($1/11 \times 3$). However, this is an optimistic assumption. For Interval 2, warp 3 cannot issue instructions 4 and 5 during the stall cycles of warp 1. To issue those instructions, extra issue cycles are required. In Section IV-A, we will illustrate and model two widely used scheduling polices: round-robin (RR) and greedy-then-oldest (GTO) policies to consider the extra issue cycles incurred in both polices.

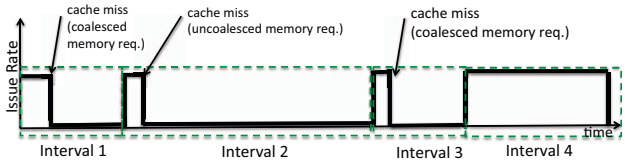



Figure 3. Interval analysis with different degrees of memory divergences.

**Resource Contention in the Memory System**: The second issue of the naïve approach is that it ignores resource contention in the memory system, which is likely to occur with multithreaded architectures. One of the unique features of a GPU architecture is that the number of memory requests from a SIMD/SIMT instruction varies significantly depending on the degree of *memory divergence*, referred to as uncoalesced memory accesses. Figure 3 illustrates that stall cycles can vary significantly because of queuing delays caused by memory divergence. To address the problem, we model the queuing delays of limited MSHR entries and DRAM bandwidth, shown in Sections IV-B1 and IV-B2.

**A Case Study**: To provide a more concrete example, we show how modeling different components reduces the error gradually. Figure 4 shows the error of interval analysis compared with a detailed timing simulation for an SRAD kernel that has divergent memory accesses from Rodinia [8].

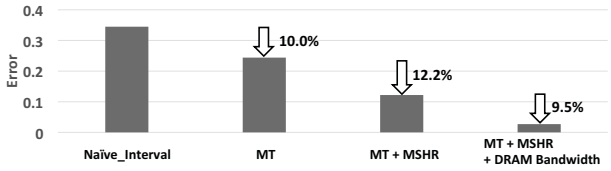



Figure 4. The errors of a kernel from the SRAD benchmark

`Naive_Interval` bar shows the error when using Eq. 1. `MT` models the round-robin scheduling policy while `MSHR` and `DRAM Bandwidth` model resource contention for the MSHRs and DRAM bandwidth, respectively. As the result shows, modeling both scheduling policy and resource contention is critical to improve the accuracy of the interval analysis technique for GPUs.

## III. Single-Warp Model

### A. GPUMech Overview

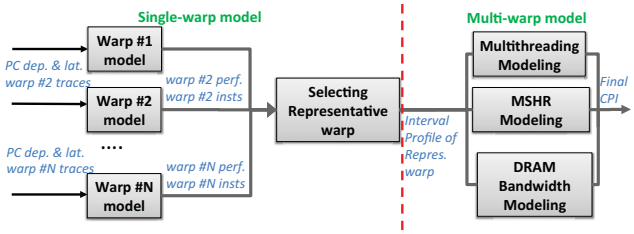



Figure 5. GPUMech Overview

Figure 5 shows the performance model of GPUMech, including single-warp and multi-warp models. The input of the single-warp model is generated by the input collector described in Section V. For the *single-warp model*, the *interval algorithm*, described in the next subsection, models an in-order execution of a warp and forms its intervals. "*Selecting the representative warp*" chooses the representative warp that has an interval profile similar to that of the majority of warps.

Eq. 2 shows an *interval profile* of a warp, defined as a collection of intervals constructed by the interval algorithm. Each interval has the information of the number of instructions and stall cycles. The interval profile of the representative warp is sent to the multi-warp model.

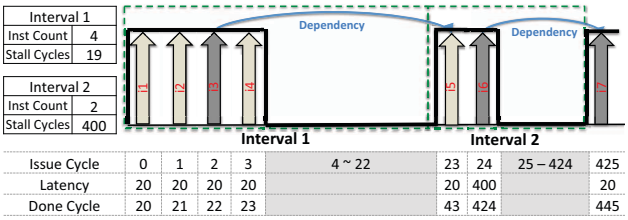
$$\text{interval\_profile} = \{[\#\text{interval\_insts}_i, \text{stall\_cycles}_i], i \in \text{intervals}\} \quad (2)$$

Eq. 3 shows the final CPI, which is the sum of the CPIs of multithreading and resource contention, predicted by the *multi-warp model*.

$$\text{CPI}_{\text{final}} = \text{CPI}_{\text{multithreading}} + \text{CPI}_{\text{rc\_contention}} \quad (3)$$

### B. Interval Algorithm

The purpose of the interval algorithm is to construct a warp's interval profile assuming an in-order execution




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Issue Cycle | 0 | 1 | 2 | 3 | 4 ~ 22 | 23 | 24 | 25 – 424 | 425 |
| Latency | 20 | 20 | 20 | 20 | | 20 | 400 | | 20 |
| Done Cycle | 20 | 21 | 22 | 23 | | 43 | 424 | | 445 |

Figure 6. Intervals of a warp. (The shaded boxes indicate the stall cycles in which no instructions are issued. The instructions in dark gray are the ones that lead to stall cycles.)

model. The inputs of the interval algorithm are (1) the instruction latency per static instruction and (2) the instruction trace of a warp tagged with dependency information. The generated interval profile of each warp will be used to select the representative warp.

Figure 6 illustrates intervals of a warp. The done cycle is equal to the issue cycle plus the instruction latency. instruction 3 (i3) leads to stall cycles because instruction 5 (i5) depends on it. On the other hand, other instructions in the first interval have no dependent instructions, so they do not cause any stall cycles.

The interval algorithm traverses every instruction in the instruction trace of a warp. It determines the issue cycle of each instruction using Eq. 4, assuming that one instruction can be issued every cycle. An interval is formed if the issue cycle of the current instruction is not equal to the issue cycle of the previous instruction plus one, since it indicates that the stall cycles are incurred between the two instructions. In that case, the previous instruction is included into the newly formed interval, while the current instruction will belong to the interval that will be formed next. The algorithm proceeds until every instruction in a warp belongs to an interval.

$$\text{issue\_cycle}(\text{inst}_{k+1}) = \max(\text{issue\_cycle}(\text{inst}_k) + 1, \text{done\_cycle}(\text{source\_inst}_{k+1}) + 1) \quad (4)$$

### C. Selecting Representative Warp

Once the interval profile of each warp is collected, we select one warp to predict the overall performance. However, as some warps may have different degrees of control divergences, their interval profiles could be quite different. Using the interval profile from a random warp as an input to the multi-warp model can lead to high error, since the warp may not be representative of the other warps. Therefore, to reduce the errors caused by control divergences, we attempt to identify the most representative warp using clustering.

Specifically, GPUMech uses the k-means algorithm, which requires two inputs: (1) the number of clusters and (2) the feature vector of each warp. We set the number of clusters to two. One cluster is to capture the majority warps with similar interval profiles while the other cluster is to

capture the outlier warps.

$$\text{warp\_perf} = \frac{\sum_{i \in \text{intervals}} (\#\text{interval\_insts}_i)}{\sum_{i \in \text{intervals}} (\frac{\#\text{interval\_insts}_i}{1.0(\text{issue rate})} + \text{stall\_cycles}_i)} \quad (5)$$

A feature vector is used to characterize the interval profile of a warp. One simple approach is to represent each interval as one dimension of the vector. Then the number of dimensions is equal to the number of intervals of a warp. However, clustering the vectors is not scalable, since a feature vector for a long-running warp may have thousands or more dimensions (intervals). Instead, we use the **warp performance**, which is the IPC when a warp is running alone on a core, as shown in Eq. 5, as one dimension of the vector. Our assumption is that if two warps have different interval profiles, their warp performances are different, and vice versa. However, the warp performance may not capture the warps with different number of intervals, e.g., distinct iteration counts, even though their performances are the same. Thus, **the number of instructions** of a warp is also used as another dimension of the vector. The final feature vector of a warp is shown in Eq. 6. Both features are normalized by the respective average value over all warps.

$$\text{feature\_vector}_w = [\frac{\text{warp\_perf}_w}{\text{avg\_warp\_perf}}, \frac{\#\text{warp\_insts}_w}{\text{avg\_warp\_insts}}], w \in \text{warps} \quad (6)$$

The representative warp is selected as the warp closest to the center of the largest cluster, since its interval profile is more likely to be representative of the majority of warps and thus will be used as the input of the multi-warp model.

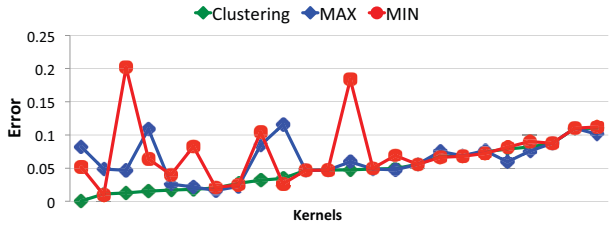



Figure 7. Errors from different representative warp selection methods. Each tick represents a control divergent kernel and data points are sorted by the errors of Clustering approach.

Figure 7 shows the errors of GPUMech when it uses three different methods to select a representative warp. (The baseline configuration is shown in Table I). The three selection methods are (1) MAX: selecting the warp with the maximum warp performance; (2) MIN: selecting the warp with the minimum warp performance and (3) Clustering: selecting the warp using clustering. For some kernels, the three cases have similar errors, indicating that the difference of interval profiles of warps is negligible. For the others, the clustering method usually has the best accuracy.

## IV. Multi-Warp Model

The multi-warp model uses the interval profile of the representative warp and predicts the performance under multiple warps, which is the typical situation on a GPU core. Specifically, it models the performance improvement due to multithreading and the degradation due to resource contention.

### A. Modeling Multithreading

We model multithreading by assuming that multiple warps are running on a core without resource contention. The key idea is to count the number of instructions of the remaining warps that hide the stall cycles of the representative warp. Later, we add resource contention modeling. Ideally, all instructions in the remaining warps hide the stall cycles. However, some instructions do not hide the stall cycles, leading to sub-optimal performance. In the following, we illustrate and model two popular warp scheduling polices: round-robin (RR) and greedy-then-oldest (GTO) [23] polices.

In our terminology, among the warps assigned to a given core, the warps other than the representative warp are referred to as the **remaining warps**. From the remaining warps, the instructions that hide the stall cycles of the representative warp are referred to as **overlapped instructions** while the other instructions are called **non-overlapped instructions**. The "intervals" and "stall cycles" are referred to as those of the representative warp, unless otherwise specified. We also differentiate between "schedule" and "issue". A warp is scheduled means that a warp is selected by the scheduler to issue instructions. But it may not be able to issue due to stalls. In this case, the next candidate warp is scheduled in the same cycle until the warp that can issue instructions is scheduled.

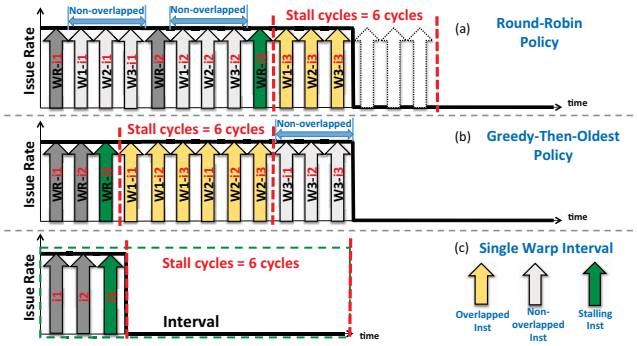



Figure 8. The cases of non-overlapped instructions of RR and GTO policies. (WR: representative warp.)

*1) The cases of non-overlapped instructions:* Figure 8 illustrates how non-overlapped instructions are incurred. To begin with, we assume that four warps are running on a core with the issue rate equal to 1 instruction per cycle, and they all have the same interval profile. For simplicity, we use one interval that has 3 instructions and 6 stall cycles, as shown in Figure 8(c).

Figure 8(a) illustrates the case for the round-robin scheduling policy, which issues an instruction every cycle in round-robin fashion. Since the instructions are issued

*regardless whether the representative warp is stalled or not*, some instructions from the the remaining warps do not hide the stall cycles. As shown in the example, three instructions from the remaining warps hide the stall cycles of the representative warp while the other instructions are interleaved with the first two instructions from the warp. The number of overlapped instructions is 3 (the instructions after WR-i3, i.e., W1-i3, W2-i3, W3-i3) while the number of non-overlapped instructions is 6 (W1-i1, W2-i1, W3-i1, W1-i2, W2-i2, W3-i2). The remaining stall cycles is 3 (the number of stall cycles minus the number of overlapped instructions). By contrast, the remaining stall cycles of the naïve prediction is 0 since all instructions from the remaining warps overlap with the stall cycles ($9 = 3 \times 3 > 6$).

Figure 8(b) illustrates the case for the greedy-then-oldest scheduling policy, which issues instructions from the same warp until it stalls. Then, the warp that has the oldest instruction is issued next and so on. As shown in the example, all instructions from W1 and W2 are overlapped with the stall cycles. However, the non-overlapped instructions are still incurred since 3 instructions (W3-i1, W3-i2, W3-i3) are not overlapped with any stall cycles. Ideally, those instructions could potentially hide the stall cycles of the next interval. But in this case, since W3 has the oldest instruction, the representative warp has to wait even though it is ready to issue.

With consideration of non-overlapped instructions, Eq. 7 shows the calculation of the multithreading CPI. The non-overlapped instructions become extra cycles added to the total cycles of the representative warp (*total_cycles*) since they do not overlap with the stall cycles. For the ease of explanation, we again assume that the issue rate is 1 instruction/cycle and the warp scheduler can issue an instruction from a warp every cycle.

$$\text{CPI}_{\text{multithreading}} = \frac{\#\text{warps} \times \sum_{\text{i} \in \text{intervals}} \#\text{interval\_insts}_\text{i}}{\text{total\_cycles} + \frac{\#\text{total\_nonoverlapped\_insts}}{1.0(\text{issue rate})}} \quad (7)$$

Eq. 8 shows how the total non-overlapped instructions is counted. The basic idea is to aggregate per-interval non-overlapped instructions, which are calculated probabilistically based on the scheduling policy. The probabilistic counting takes into account cases in which the intervals of warps could randomly interleave while Figure 8 illustrates the case where the warps are well aligned.

$$\#\text{total\_nonoverlapped\_insts} = \sum_{\text{i} \in \text{intervals}} \#\text{nonoverlapped\_insts}_\text{i} \quad (8)$$

Before modeling the non-overlapped instructions for a given scheduling policy, we first define the **issue probability** as the probability that a warp can issue an instruction in a cycle, as shown in Eq. 9. Issue probability is computed with a representative warp, and we assume that all other remain warps have the same uniform distribution.

$$\text{issue\_prob} = \frac{\sum_{\text{i} \in \text{intervals}} (\#\text{interval\_insts}_\text{i})}{\sum_{\text{i} \in \text{intervals}} (\frac{\#\text{interval\_insts}_\text{i}}{1.0(\text{issue rate})} + \text{stall\_cycles}_\text{i})} \quad (9)$$

Next, we explain how to estimate the number of non-overlapped instructions of an interval when a scheduling policy, round-robin or greedy-then-oldest, is used.

*2) Modeling Round-Robin Policy:* To model the non-overlapped instructions in the round-robin policy, we identify the instructions that are issued within several "waiting slots", where a "waiting slot" is the time period between scheduling two instructions from the representative warp during an interval. For example, in Figure 8(a), there are two waiting slots, one between WR-i1 and WR-i2 and another between WR-i2 and WR-i3. Eq. 10 shows the number of waiting slots of an interval i.

$$\#\text{waiting\_slots}_\text{i} = \#\text{interval\_insts}_\text{i} - 1, \text{i} \in \text{intervals} \quad (10)$$

Within a waiting slot, every remaining warp is scheduled because of the round-robin scheduling. Eq. 11 shows the expected number of instructions issued from all remaining warps within all waiting slots of interval i, which is equal to the number of non-overlapped instructions of the interval.

$$\#\text{nonoverlapped\_insts}_\text{i} = \text{issue\_prob} \times (\#\text{warps} - 1) \times \#\text{waiting\_slots}_\text{i} \quad (11)$$

Finally, the CPI of multithreading can be calculated using Eq. 8 and 7.

*3) Modeling Greedy-Then-Oldest Policy:* To model the non-overlapped instructions in the greedy-then-oldest policy, we identify the instructions that are issued after the stall cycles of an interval are fully overlapped. To count the number of non-overlapped instructions of the interval i, we first need to count the total number of instructions issued before the representative warp is re-scheduled, as shown in Eq. 12. This number is estimated as the multiplication of the number of issued instructions for a remaining warp (*#avg_interval_insts*) and the number of warps issued before the representative warp is re-scheduled (*#issue_warps_in_stall*$_i$). Both terms are explained in Eq. 13 and Eq. 14.

$$\#\text{issue\_insts\_in\_stall}_\text{i} = \text{avg\_interval\_insts} \times \#\text{issue\_warps\_in\_stall}_\text{i} \quad (12)$$

The number of issued instructions of a remaining warp is the number of instructions in the interval that it is currently executing. Since this is unknown information, we instead estimate that value by taking the average number of instructions of an interval as the number of issued instructions of a remaining warp, as shown in Eq. 13.

$$\text{avg\_interval\_insts} = \sum_{\text{i} \in \text{intervals}} \frac{\#\text{interval\_insts}_\text{i}}{\#\text{intervals}} \quad (13)$$

Eq. 14 shows the number of warps that issues instructions during the stall cycles of interval i. The number of remaining warps (*#warps − 1*) is multiplied based on the assumption that during the stall cycles, every remaining warp will be

scheduled since the "oldest" policy equalizes the probability of each warp being scheduled to prevent starvation.

$$\#\text{issue\_warps\_in\_stall}_i = \text{issue\_prob\_in\_stall}_i \times (\#\text{warps} - 1) \quad (14)$$

Eq. 15 shows the issue probability of a remaining warp during the stall cycles. Recall that we assume an uniform distribution of the issue probability.

$$\text{issue\_prob\_in\_stall}_i = \max(\text{issue\_prob} \times \text{stall\_cycles}_i, 1) \quad (15)$$

Finally, Eq. 16 shows the number of non-overlapped instructions of interval i by subtracting the number of stall cycles from the total issued instructions (Eq. 12). The non-overlapped instructions are incurred if the number of issued instructions is more than the stall cycles.

$$\#\text{nonoverlapped\_insts}_i = \min(\#\text{issue\_insts\_in\_stall}_i - \text{stall\_cycles}_i \times 1.0(\text{issue rate}), 0) \quad (16)$$

Similar to the round-robin policy, the CPI of multithreading can be calculated using Eq. 8 and 7.

### B. Modeling Resource Contention

One of the most prominent cause of resource contentions in GPUs is *memory divergence* (a.k.a. uncoalesced memory accesses). We do not model the resource contention for normal operations by assuming that *in a balanced GPU design, the resources used for normal operations are sufficient for each warp.* For example, if the number of threads is 32 in a warp, then the floating-point units should be at least 32 (or 16 if a warp takes two cycles to issue) to make all threads progress equally.

To model the queuing delays caused by memory divergence, different scheduling policies may affect the queuing delays as the instruction orderings are different. However, we find that the instruction orderings matters only when the degree of contention is low. Otherwise, different instruction orderings cannot change the queuing delays much because the incurred queuing delays are far more than any instruction ordering can overlap. Thus, our resource contention models, described below, are applied to both round-robin and greedy-then-oldest policies.

Specifically, we model the queuing cycles caused by the contention for (1) MSHRs and (2) DRAM bandwidth. Similar to the multithreading model, we leverage the interval profile of the representative warp to predict the queuing delays. Eq. 17 shows the queuing delay calculated per-interval basis.

$$\text{CPI}_{\text{rc\_contention}} = \frac{\sum_{i \in \text{intervals}} (\text{MSHR\_delay}_i + \text{Bandwidth\_delay}_i)}{\sum_{i \in \text{intervals}} \#\text{interval\_insts}_i} \quad (17)$$

*1) Modeling MSHRs:* Figure 9 illustrates the queuing delays resulting from a limited number of MSHRs. Assume that the number of MSHR entries are 6 and all warps have the same interval profiles containing two compute instructions and one load instruction. The load instruction issues two memory requests and which both result in an L1 cache miss. The MSHR entries are saturated after three warps issue (W1-W3) the load instructions. Thus, W4 incurs queuing delays before its load instruction can be issued.

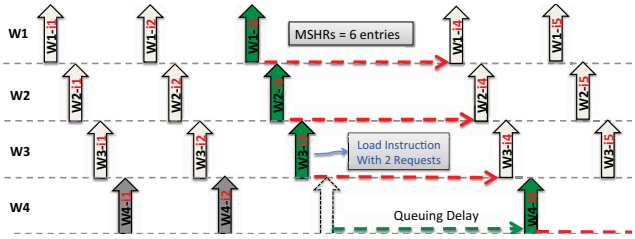



Figure 9. The queuing delays caused by a limited number of MSHR.

Since the queuing delay of a memory instruction varies depending on the number of the remaining warps issuing memory requests beforehand, we probabilistically estimate the queuing delay of every interval of the representative warp. First, we estimate the number of concurrent memory requests of a core in an interval ($\#core\_reqs_i$). Second, we calculate the expected queuing delay of a memory request ($exp\_queuing\_delay_i$). Finally, by multiplying the expected queuing delay with the number of memory instructions in an interval, we get the queuing delay of the interval due to limited MSHR entries ($MSHR\_delay_i$). Details are as follows.

Eq. 18 shows the number of concurrent memory requests of a core in interval i. For each interval, we count the expected number of concurrent memory requests as the total requests issued from all warps in a core.

$$\#\text{core\_reqs}_i = \#\text{warp\_mem\_reqs}_i \times \#\text{warps}, i \in \text{intervals} \quad (18)$$

Eq. 19 shows the expected latency of a memory request in interval i considering a limited number of MSHRs. *avg_miss_latency* is the average L2/DRAM access latency of all memory instructions without MSHR contention. Since the distribution of the L1/L2 miss rates of a memory instruction can be calculated from the input collector (Section V-B), we take the average L2/DRAM access latency across all memory instructions to get *avg_miss_latency*. Then, we estimate the latency of a memory request with index j in MSHRs as "$avg\_miss\_latency \times \left\lceil \frac{j}{\#MSHR} \right\rceil$". For example, in Figure 9, the latency of the memory requests from W1, W2 and W3 is *avg_miss_latency* while the latency of those from W4 is $avg\_miss\_latency \times 2$ since the memory requests from W1, W2 and W3 can be serviced concurrently while W4 has to wait until the MSHR entries are freed. By taking the average, we get the expected latency of a memory request. Finally, by subtracting *avg_miss_latency* from the expected miss latency, we get the expected queuing delay of a memory request caused by limited MSHRs.

$$\text{exp\_queuing\_delay}_i = \frac{\sum_{j=1}^{\#\text{core\_reqs}_i} \text{avg\_miss\_latency} \times \left\lceil \frac{j}{\#\text{MSHR}} \right\rceil}{\#\text{core\_reqs}_i} - \text{avg\_miss\_latency} \quad (19)$$

Eq. 20 shows the queuing delay of interval i in a core due to MSHR contention. The queuing delay only occurs when the expected number of memory requests exceeds the number of MSHRs. Note that *#warp_mem_insts$_i$* represents the number of memory instructions in interval i from a warp. We count the queuing delay *per-memory instruction* not *per-memory request* since a divergent memory instruction has multiple memory requests issued concurrently, thereby overlapping the queuing delay.

$$\text{MSHR\_delay}_i = \begin{cases} 0, & \#\text{core\_reqs}_i \leq \#\text{MSHR} \\ \text{exp\_queuing\_delay}_i & \\ \times \#\text{warp\_mem\_insts}_i, & \#\text{core\_reqs}_i > \#\text{MSHR} \end{cases} \quad (20)$$

In Eq. 19, we assume that all warps issue the memory instructions of an interval at the same cycle. However, in reality, memory requests are issued at different cycles depending on how warps are scheduled. In that case, some warps may incur less queuing delay if they issue the memory instructions at a later time. As the difference of the progresses of different warps tend to be much smaller than the incurred stall cycles, the difference of queuing delay between warps is ignored.

The approach used to model queuing delays can be generalized to model other components with resource contention problems, such as the special functional unit (SFU). Applying the approach for these components is left for the future work.

*2) Modeling DRAM Bandwidth:* As a large number of memory requests from the warps are likely to be issued within a short amount of time, limited DRAM bandwidth is another major bottleneck in the memory system. However, the queuing delay model of MSHRs cannot be applied to the DRAM queue since the DRAM queue has a much shorter service time. For example, the service time of a memory request in MSHRs is the miss latency of the request while that in the DRAM queue is the transmission time of a cache line on the DRAM bus. In the case of short service times, estimating the arrival time between different requests is crucial since it can affect the queuing delay.

To approximate the arrival time of each request, we leverage M/D/1 queue, an approach used to model queuing delays in parallel simulations [21], as shown in Eq. 21. It states that the arrival rate follows a Poisson process and the service time is constant. The expected DRAM queuing delay of interval i composed of the utilization ($\rho$) and the arrival rate ($\lambda$). However, for some intervals with a large number of memory requests, $\rho$ could be equal to 1 resulting in unlimited queuing delay. To prevent this, we cap the queuing delay by assuming that a request arrives at the queue in which half of the maximum number of requests are ahead of it ($\frac{\#core\_reqs_i \times \#cores}{2}$).

$$\text{Bandwidth\_delay}_i = \min\left(\frac{\lambda_i s^2}{2(1-\rho)}, s \times \frac{\#\text{core\_reqs}_i \times \#\text{cores}}{2}\right) \quad (21)$$

Eq. 22 shows the utilization $\rho$ in which the service time $s$ is represented in cycles as $freq_{core} \times \frac{L}{B}$ while the service time of the request in the DRAM queue is $\frac{L}{B}$ where L is the cache line size and B is the DRAM bandwidth.

$$\rho(\text{utilization}) = \lambda s = \lambda(\text{freq}_{\text{core}} \times \frac{L}{B}) \quad (22)$$

Eq. 23 shows the aggregated arrival rate from all cores. To estimate the arrival rate $\lambda$, we assume that the memory requests of a given interval from all warps can be issued within the total cycles of the interval ($\frac{\#interval\_insts_i}{1.0(issue\ rate)} + stall\_cycles_i$).

$$\lambda_i(\text{arrival\_rate}) = \frac{\#\text{core\_reqs}_i \times \#\text{cores}}{\frac{\#\text{interval\_insts}_i}{1.0(\text{issue rate})} + \text{stall\_cycles}_i} \quad (23)$$

## V. Input Collector

### A. Overview

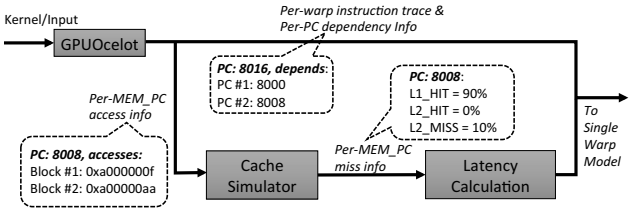



Figure 10. The input collector

Figure 10 shows the input collector of GPUMech. GPUOcelot [11] is used as a functional simulator, which executes a GPGPU kernel and generates *per-warp* instruction traces with dependency information. The cache simulator reads the memory instructions and their addresses from the trace of each warp in a round-robin fashion to get the *per-memory PC*[1] miss rates. The cache simulator models a system with the number of warps and cores equal to that of the modeled system without timing information. Once the latency of a PC is determined (Section V-B), both per-warp instruction trace and the per-PC latency are used as the inputs to the single-warp model.

### B. Instruction Latency per PC

To determine the latency of per PC, we classify the PCs into *compute* and *memory* PCs. The latencies of compute PCs are fixed and based on the system configurations. On the other hand, the latencies of memory PCs are determined as follows.

First, we simulate L1/L2 caches to collect the *distribution of miss events* for every memory PC, e.g., (L1 hit: 0%,

[1]We use the term PC to indicate a static instruction.

L2 hit: 10%, L2 miss: 90%). For a divergent memory instruction, it may have some cache hits at different levels of the memory hierarchy. In this case, the miss event of the memory instruction is determined by the memory request with the longest latency. Second, the distribution of miss events is collected by counting the miss events of every memory instruction across all warps. Lastly, the latency of a memory PC is equal to the *average memory access time* (AMAT) of the PC. For example, a memory PC hits L2 cache (120 cycles) for 90% of the executions and misses L2 cache (420 cycles) for 10% of the executions, the latency of the memory PC is equal to $150 = 0.9 \times 120 + 0.1 \times 420$ cycles.

In this work, we do not model front-end stall events, such as I-cache misses and synchronization overhead. Because warps share the same kernel (SIMT programming model), the I-cache miss rate is very low for all kernels. In most kernels, synchronization usually occurs occasionally within a thread block, e.g. by calling synchthreads(). Since the warps in a thread block are likely to make similar progress, the within-thread-block synchronization overhead is typically low. Synchronization across thread blocks is rarely used in GPGPU kernels since the operation requires atomic instructions, which cause significant slowdown. Please note that control divergence is not considered as one of the stall events since it does not stall the entire warp (some threads in a warp still make progress).

## VI. Evaluation

### A. Methodology

We use Macsim [20], a cycle-level simulator to validate GPUMech. The simulation configurations are listed in Table I. The latency of the network-on-chip is included as part of the L2 latency. The validation is done by calculating the *relative error* of the CPI predicted by the evaluated model with that of detailed timing simulation.

To demonstrate the wide applicability of our model, we evaluated the kernels from Rodinia 2.1 [8], Parboil 2.5 suites [26] and NVIDIA SDK (40 kernels in total). The evaluated kernels have at least $3 \times$ system_occupancy thread blocks to have enough length of simulation.

Table II shows the evaluated models. `Naive_Interval` is the same as Eq. 1 in Section II-B. In addition to evaluating the relevance of modeling different components, we compare our results with a Markov Chain that models the multithreaded performance in [9]. The Markov Chain model is discussed in Section VIII.

### B. Model Accuracies of Different Scheduling Policies

Figure 11 shows the comparisons between different models using the round-robin policy. `Markov_Chain` shows a slightly better result than the `Naive_Interval` since it models the multithreading effect more accurately. However, compared to `MT`, it still overestimates the performance. On the other hand, as `MT` does not account for the resource contention, some kernels with high memory divergence

Table I
Simulation Configuration.

| | |
|---|---|
| Number of cores | 16 |
| Front End | Fetch width: 1 warp-instruction/cycle,<br>4KB I-cache |
| Execution core | 1.0 GHz,<br>SIMT width: 32,<br>Warp size: 32 threads,<br>Maximum threads: 1024 threads,<br>Issue width: 1 warp-instruction/cycle,<br>Instruction latencies are modeled according to<br>the CUDA manual (normal FP instructions are 25 cycles) |
| On-chip caches | 16 KB software managed cache<br>32 KB L1 cache, 128B line, 25 cycles,<br>8-way assoc, 32 MSHR entries<br>768 KB L2 cache, 128B line, 120 cycles 8-way assoc |
| DRAM | DRAM Bandwidth: 192 GB/s<br>Access latencies: 300 cycles |

Table II
Evaluated Models.

| Evaluated Models | Descriptions |
|---|---|
| `Naive_Interval` | Optimistic overlap |
| `Markov_Chain` | Markov chain based model [9] |
| `MT` | Modeling multithreading (Section IV-A) |
| `MT_MSHR` | Modeling Multithreading + MSHR (Section IV-B1) |
| `MT_MSHR_BAND` **(GPUMech)** | Modeling Multithreading + MSHR<br>+ DRAM Bandwidth (Section IV-B2) |

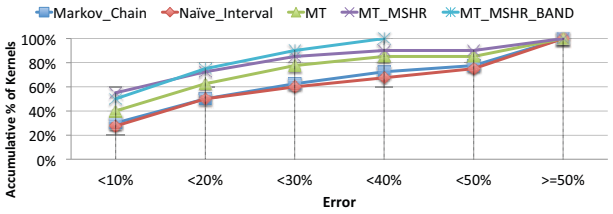



Figure 11. Model comparisons for round-robin policy

have high errors, such as *invert_mapping* from the *kmeans* benchmark that has a 330% error. `MT_MSHR` reduces the errors for most kernels down to less than 30%, except for the kernel *invert_mapping* from the *kmeans* benchmark. The reason why `MT_MSHR` is effective for most cases without modeling DRAM bandwidth is that the number of MSHR entries caps the number of read requests, making the queuing delay of DRAM bandwidth bounded. However, since write requests do not allocate MSHR entries and *invert_mapping* has a large number of divergent write requests, the queuing delay of DRAM bandwidth is significant and has to be modeled. `MT_MSHR_BAND` (GPUMech) further reduces the error of *invert_mapping* from 180% to 35%. The remaining error is mainly caused by the inherit errors from modeling using queuing theory. Overall with our proposed mechanism GPUMech, 75% of the kernels have less than 20% errors, while only 50% of kernels have less than 20% errors with `Markov_Chain`. Furthermore, 32 warp case in Figure 13 (our baseline), shows that the average error of GPUMech is 13.2% while the `Markov_Chain` average error is 62.9%.

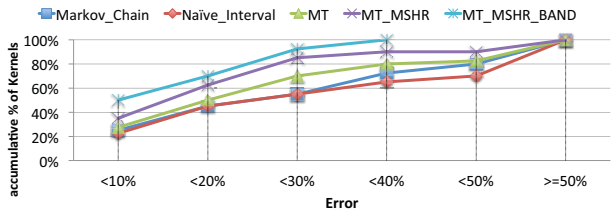



Figure 12. Model comparisons for greedy-then-oldest policy

Figure 12 shows the comparisons between different models using the greedy-then-oldest policy. The average error of GPUMech is 14.0% while the `Markov_Chain` error is 65.3%, showing a similar trend as modeling the round-robin policy. Overall, GPUMech models both scheduling policies with high accuracy.

### C. Varying Hardware Configurations

In order to demonstrate the robustness of our model, we varied the number of warps per core, the number of MSHR entries and the DRAM bandwidth to see the accuracy impact of our models for multithreading, MSHR and DRAM bandwidth. The error in the Y-axis is the relative error *averaged over* all kernels. Since both round-robin and greedy-then-oldest policies show a similar trend, in the following figures, we report the results of modeling the round-robin policy only.

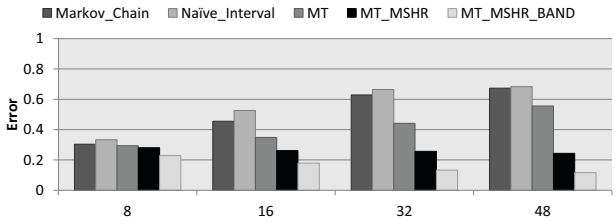



Figure 13. Errors with different number of warps per core

Figure 13 shows the errors with different number of warps. For the importance of resource contention modeling, except for `MT_MSHR` and `MT_MSHR_BAND`, the errors of other models increase with the number of warps per core since the delays caused by memory contentions get more severe with more warps. Although the MSHR is rarely congested when the number of warps is 8, some kernels still have significant queuing delays in DRAM bandwidth because of the write traffic, such as the two kernels from the *sad* benchmark. For the importance of multithreading modeling, when the number of warps is low, the prediction accuracies between `MT`, `Markov_Chain` and `Naive_Interval` are similar since the probability that instructions overlap with stall cycles is high. However, when the number of warps increases, the percentage of non-overlapped instructions increases. In this case, `MT` has a better accuracy than `Markov_Chain` and `Naive_Interval`. Overall, in GPUMech, the errors are higher in low number of warps per core, because it has more variations of multithreading. However, all other modeling techniques show higher errors as the number of warps per core increases, which makes GPUMech more attractive.

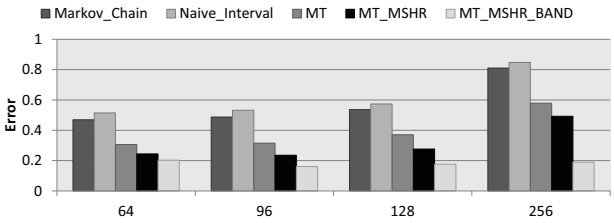



Figure 14. Errors with different number of MSHR entries

Figure 14 shows the errors with different number of MSHR entries. Increasing the number of MSHR entries increases the importance of modeling DRAM bandwidth. With more MSHR entries, the queuing delays in MSHR decrease, so the error differences between `MT` and `MT_MSHR` also decrease. However, more MSHR entries increase the queuing delays of DRAM since more on-the-fly memory requests need to be consumed. Only `MT_MSHR_BAND` captures these increases in DRAM congestion well since with all other modeling techniques, error increases as the number of MSHR entries increases.

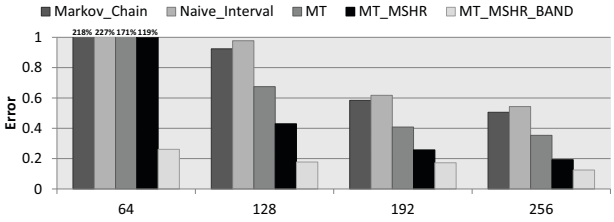



Figure 15. Errors with different DRAM bandwidth (the unit of X-axis is GB/s)

Figure 15 shows the errors with different DRAM bandwidth. DRAM bandwidth modeling is more important when the DRAM bandwidth is lower since lower DRAM bandwidth indicates higher DRAM queuing delays. The difference between modeling `MT_MSHR_BAND` and other models becomes smaller as the DRAM bandwidth increases. When the DRAM bandwidth is low, the errors are higher even with `MT_MSHR_BAND` indicating that it requires a more accurate determination of DRAM queuing delays. For 64 GB/s, GPUMech shows 26.1% error, while for all other configurations the error was less than 17.8%.

### D. Discussions on Timing Overhead

In this section, we discuss the timing overhead of GPUMech compared to a detailed timing simulator.

GPUMech has three areas of overhead: *warp clustering*, *interval algorithm* and *cache simulation*. Since GPUOcelot

is used for collecting traces for both detailed simulation and GPUMech, its overhead is excluded. First, the overhead of warp clustering could be significant depending on how many warps a kernel has. However, clustering only incurs a one-time cost in the per-input-basis. For the kernel with 100,000 warps, the clustering overhead is a few seconds. Second, the instruction algorithm is several hundred times faster than detailed simulations. In addition, the speedup can be further increased by running the interval algorithm of each warp in parallel, but we did not explore this option. Third, on average, the cache simulator is around 108x faster than our detailed simulator. Overall, GPUMech is 97x faster than detailed simulation.

To model a different hardware configuration, since the stall cycles may have changed, *cache simulation* and *the interval algorithm* of the representative warp need to be rerun. Because the representative warp is already selected, running interval algorithms on the remaining warps and warp clustering are not needed as these tasks are applied in the per-input-basis and remain unchanged across different hardware configurations. Overall, the speedup would be higher when exploring different hardware configurations since the profiling cost can be further reduced by excluding the cost of warp clustering and the running of the interval algorithms on the remaining warps.

## VII. Application

An advantage of GPUMech is the ability to construct CPI stacks which are used to visualize the performance bottlenecks and their relative impact. Table III shows the total categories in which cycles are spent.[2] We construct the CPI stack of a kernel as follows.

- We construct the CPI stack of the representative warp. To determine the CPI category (excluding `BASE`[3]), we check the reason for stalling. If the stall cycles are caused by a dependence on a compute instruction, we add the cycles to `DEP` category. If the stall cycles are caused by a memory instruction, we use the distribution of miss events to categorize the cycles. For example, if the stall cycles are 100 while the distribution of miss events is L2 hit: 10% and L2 miss: 90%, we then add 10 cycles to `L2` category while we add 90 cycles to `DRAM` category assuming no MSHR and DRAM queuing delays. By dividing each category with the number of instructions, we get the CPI stack of the representative warp. After the category of an interval is determined, we add the stall cycles of the interval to the category.
- Based on the performance improvement of multithreading, we shrink down each category of the CPI stack of the representative warp by the factor of $\frac{\text{CPI}_{\text{multithreading}}}{\text{CPI}_{\text{repres\_warp}}}$. By doing this, the relative importance of each category is reserved while modeling the performance under multithreading.
- To model the resource contention, we create two new categories: `MSHR` and `QUEUE`. The modeled queuing delays of MSHR and DRAM bandwidth are normalized by the number of instructions before being added to `MSHR` and `QUEUE` categories.

[2]Note that the DRAM access latency is the base DRAM access latency (300 cycles in our case) without queuing delays

[3]`BASE` category is the cycles used to issue an instruction, which is a constant depending on the system configuration.

Table III
Stall types of CPI stacks.

| Stall types | Abbreviations |
|---|---|
| Instruction issue cycles | `BASE` |
| Compute Dependencies | `DEP` |
| L1 Hits | `L1` |
| L2 Hits | `L2` |
| DRAM access latency | `DRAM` |
| MSHR queuing delay | `MSHR` |
| DRAM queuing delay | `QUEUE` |

### A. Identify the Scaling Bottlenecks

In this application, we leverage the CPI stack to visualize the performance bottlenecks with varying the number of warps on a core. Increasing the number of warps may reduce the cycles spent in `DEP` because the computation takes fixed cycles while more warps can reduce the stall cycles caused by computation dependencies. However, the cycles spent in `L1` and `L2` depend on the miss rates where more warps may cause higher miss rates due to cache pollution. On the other hand, the cycles in `MSHR` and `QUEUE` may increase since more warps compete for those shared resources. GPUMech is able to visualize performance bottlenecks using the CPI stack and then find the configuration that has the best performance.

We select three kernels with distinct memory divergence degrees from Rodinia suite [7]. cfd_step_factor kernel is a coalesced kernel with no divergent accesses. cfd_compute_flux has medium range of divergence since some memory instructions have up to 16 diverged requests. kmeans_invert_mapping has maximum memory divergence with up to 32 diverged requests per memory instruction.

Figure 16 illustrates the CPI stacks, shown in bars. The lines show the *oracle CPI* measured using detailed simulations. All CPIs are normalized by the oracle CPI of the configuration with 8 warps. As seen in the figure, GPUMech accurately predicts the scaling trend. In the following, we explain the performance limiting factors for each kernel.

- **cfd_step_factor**: Because all memory accesses are almost coalesced, the cycle overhead from `MSHR` and `QUEUE` are negligible in all configurations except for the 48 warps configuration, in which case larger MSHR queuing delays are incurred. Even though the major performance bottleneck is DRAM accesses, cfd_step_factor has a good scaling capability since the accesses are well spread into different intervals without congesting the memory system. In addition, we can see that cfd_step_factor has negligible

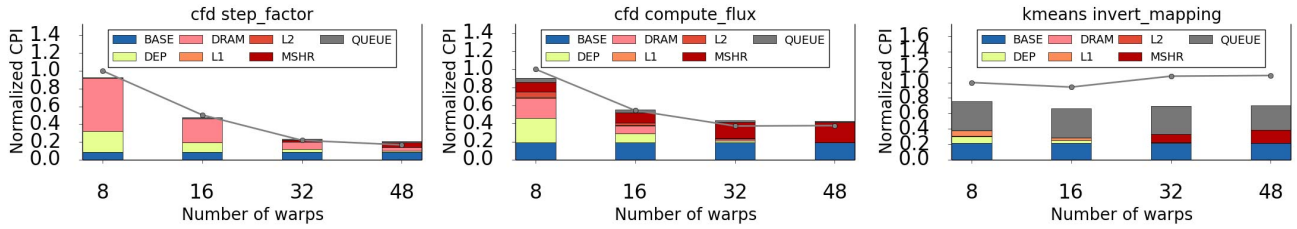



Figure 16. The CPI stacks of cfd_step_factor, cfd_compute_flux and kmeans_invert_mapping kernels.

or no cache locality since no cycles are seen in the `L1` or `L2` categories.

- **cfd_compute_flux**: In the configuration of 8 warps, `DRAM` and `DEP` have a similar amount of cycle overhead. The L2 cache is effective for the diverged accesses while L1 is not very useful since the working set is much larger than the L1 size. In terms of memory congestion, `MSHR` contributes to 10.8% of the cycle overhead while `QUEUE` contributes to 3.8% of the cycle overhead, thanks to the high degree of memory divergences. Since `MSHR` and `QUEUE` are not the majority of the cycle overhead, increasing the number of warps improves performance, as shown from the configurations of 8 to 16 warps. The predicted performance saturation point occurs in the configuration of 32 warps in which `MSHR` dominates the cycle overhead.
- **kmeans_invert_mapping**: In the configuration of 8 warps, since the load instructions have a high L1 hit rate (90.5%), the cycles spent in the L1 cache is significant. The `MSHR` is negligible even though the divergence degree is high thanks to the high L1 hit rate preventing most accesses from occupying MSHR entries. On the other hand, it is a bit counter-intuitive to see that `DRAM` has negligible cycle overhead while `QUEUE` is high. The reason is that while the store instruction is not on the critical path and does not increase the cycles in `DRAM` category, the diverged write accesses still consume DRAM bandwidth and increase the queuing delays of the load instruction.

Because the performance bottlenecks could come from multiple sources, without CPI stacks, it is hard to tell what limits the performance of a given hardware configuration. By showing the relative importance of performance bottlenecks, the CPI stack is useful for software developers to apply the corresponding optimizations and for hardware developers to scale the required hardware resources, such as MSHR entries, DRAM bandwidth, to achieve the optimal performance. By leveraging the proposed model, not only can we find the performance saturation point, but also the details of performance bottlenecks.

## VIII. Related Work

### A. Analytical Model of a Multithreaded Core

Chen and Aamodt proposed a first-order performance model of a multithreaded core [9]. They performed a Markov Chain analysis to predict the performance of a multithreaded core similar to a GPU core except with vector processing. We evaluated this model in Section VI as `Markov_Chain`. To begin with, a single thread is modeled as a random variable with two states: *activated* and *suspended*. Activated means that the thread can issue an instruction at the cycle. Otherwise, it is stalled. The transition probability from activated to suspended is p while the probability from suspended to activated is $\frac{1}{M}$, where M is the number of cycles of a thread being suspended. By performing the Markov chain analysis, the probability of being in any state at any cycle can be known.

However, the model has two limitations, which are most likely the causes of high errors in our evaluation. First, the model assumes that threads are randomly interleaved without modeling any specific scheduling policy. Second, because the model assumes that a thread can issue no more than one memory request, it underestimates the queuing delays due to the resource contention in the memory system, especially for memory divergent kernels on GPUs.

### B. GPU Models

In the past few years, several GPU performance models have been proposed. Baghsorkhi et al. [2] proposed to use work flow graph (WFG), an extension from the control flow graph, to estimate the performance. Zhang and Owens [27] proposed a model to measure the execution time of the instruction pipeline, shared memory, and global memory respectively using micro-benchmarking. However, these prior works did not model the cache hierarchy that is equipped in all modern GPUs.

GPUPerf [24] is a performance analysis framework used to predict the performance bottlenecks of GPGPU kernels. They used benefit metrics to indicate the relative importance of different performance bottlenecks. The benefit metrics are generated by the performance model extended from the MWP-CWP model [16]. But, similar to the model for a multithreaded core, they do not model the queuing delay due to resource contention in the memory system nor do they model the scheduling policy. This does not only affect the modeling accuracy, but also the reported benefit metrics. For example, the model may suggest to increase the memory-level parallelism (MLP), but it may instead hurt the performance due to increased queuing delays. In addition,

the proposed benefit metrics *do not show what limits the performance and by how much.*

In addition to performance models, several GPU cache models have been proposed. Baghsorkhi et al. [3] applied the Monte Carlo simulation for a finite number of times to mimic the non-deterministic schedule deviation between thread blocks. Tang et al. [25] applied the reuse distance theory on a single thread block to model the cache miss rate without considering MSHRs. Nugteren et al. [22] proposed a cache model for L1 cache based on the reuse distance theory. They emulate per-warp memory traces with the round-robin scheduling policy. In addition, they modeled MSHRs accounting for a limited number of outstanding misses.

### C. Methods based on Interval Analysis

Several studies improved or applied the interval analysis technique. Genbrugge et al. [15]. proposed "interval simulation", which improves the simulation speed by abstracting the out-of-order execution using interval analysis. Eyerman et al. [14], [12] proposed performance counter architectures for out-of-order processors and SMT processors based on interval analysis. Chen et al. [10] improved the accuracy of the technique considering pending cache hits, prefetching and MSHRs. Breughe et al. [5] applied the technique to in-order processors.

## IX. Conclusion

In this work, we proposed GPUMech, which is the first interval analysis for GPU architectures. GPUMech models multithreading and resource contentions in MSHR and DRAM bandwidth due to memory divergence. To reduce the errors from control-divergent warps, we employ a clustering algorithm to identify representative warps. Overall, GPUMech is about 97x faster than a detailed timing simulator and on average, it only has 13.2% error for modeling round-robin scheduling policy and 14.0% error for modeling greedy-then-oldest policy. In addition, GPUMech generates CPI stacks, which helps the hardware/software developers to visualize performance bottlenecks.

## Acknowledgment

We gratefully acknowledge the support of National Science Foundation (NSF) CAREER CCF 1054830. We would like to thank Dilan Manatunga and other HPArch members and the anonymous reviewers for their comments and suggestions. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF.

## References

[1] J. Andrews and C. Polychronopoulos. An analytical approach to performance/cost modeling of parallel computers. *JPDC '91*.

[2] S. Baghsorkhi, M. Delahaye, S. Patel, W. Gropp, and W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP '10*.

[3] S. Baghsorkhi, I. Gelado, M. Delahaye, and W. Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *PPoPP '12*.

[4] G. Bernacchia and M. Papaefthymiou. Analytical macromodeling for high-level power estimation. In *ICCAD '99*.

[5] M. Breughe, S. Eyerman, and L. Eeckhout. A mechanistic performance model for superscalar in-order processors. In *ISPASS '12*.

[6] J. Butts and G. Sohi. A static power model for architects. *MICRO*, 0:191–201, 2000.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC '09*.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *JPDC '08*.

[9] X. Chen and T. Aamodt. A first-order fine-grained multithreaded throughput model. In *HPCA '09*.

[10] X. Chen and T. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. *TACO '11*.

[11] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *PACT-19*, 2010.

[12] S. Eyerman and L. Eeckhout. Per-thread cycle accounting in smt processors. In *ASPLOS '09*.

[13] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith. A mechanistic performance model for superscalar out-of-order processors. *TOCS '09*.

[14] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith. A performance counter architecture for computing accurate cpi components. In *ASPLOS '06*.

[15] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *HPCA '10*.

[16] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA '09*.

[17] W. Jia, K. Shaw, and M. Martonosi. Stargazer: Automated regression-based gpu design space exploration. In *ISPASS '12*.

[18] T. Karkhanis and J. Smith. Automated design of application specific superscalar processors: an analytical approach. In *ISCA '07*.

[19] T. Karkhanis and J. Smith. A first-order superscalar processor model. In *ISCA '04*.

[20] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho. *MacSim: A CPU-GPU Heterogeneous Simulation Framework User Guide*. Georgia Institute of Technology, 2012.

[21] J. Miller, H. Kasture, G. Kurian, C. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA '10*.

[22] C. Nugteren, G. van den Braak, H. Corporaal, and H. Bal. A detailed GPU cache model based on reuse distance theory. In *HPCA '14*.

[23] T. Rogers, M. O'Connor, and T. Aamodt. Cache-conscious wavefront scheduling. In *MICRO '12*.

[24] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *PPoPP '12*.

[25] T. Tang, X. Yang, and Y. Lin. Cache miss analysis for gpu programs based on stack distance profile. In *ICDCS '11*.

[26] The IMPACT Research Group, UIUC. Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.

[27] Y. Zhang and J. Owens. A quantitative performance analysis model for GPU architectures. In *HPCA '11*.